

REPORT

Digital Logic Circuit

3조



과목명 | 디지털 논리 회로

담당 교수 | 박 재 근 교수님

학 과 | 전자정보공학부(IT융합전공) / 신소재공학부

팀 원 | 20201580 김 철 중 | 20201597 오 예 찬 | 20211089 이 준 영

제출일 | 2024 . 06. 01 .

목 차

I. 프로젝트 개요 및 목표	3
II. 구성 요소	4
1. Opcode	4
2. MINI-CPU	7
III. 회로도 + 파이프라인	8
1. 회로도	8
2. 파이프라인	9
3. Hazard 해결 방안	10
IV. 코드	11
1. 파이프라인 단계별 모듈	11
2. 각 단계 별 코드 설명	12
2-1. Fetch 단계	12
2-2. Decode 단계	13
2-3. Execute 단계	14
2-4. Memory 단계	15
2-5. Write-back 단계	16
2-6. CPU	17

V. 수행 결과	19
VI. 프로젝트 수행일지	20
VII. Trouble shooting	21

I. 프로젝트의 개요 및 목표

- HDL(Hardware Description Language) 언어 이해
- CPU의 기본 동작 원리, 디지털 회로 기본 동작 원리 이해
- Verilog 코드를 활용하여 16비트 CPU의 간단한 ALU 동작을 구현하고, 이를 5단계 파이프라인 구조로 설계합니다.

이 프로젝트의 목표는 5단계 파이프라인 구조를 갖춘 16비트 CPU를 설계하고 구현하는 것입니다. 이를 위해 각 파이프라인 단계인 Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), Write Back (WB) 모듈을 독립적으로 설계한 후, 이를 통합하여 하나의 CPU 모듈로 구현합니다. 이러한 모듈화된 접근 방식은 각 단계의 기능을 명확하게 분리하여 CPU의 동작을 최적화하고 성능을 향상시키는 것을 목표로 합니다. 또한, Verilog 코드를 사용하여 다양한 명령어와 ALU 연산을 수행할 수 있는 16비트 CPU를 설계하며, 이를 통해 파이프라인 구조의 효율성을 극대화하는 것을 목표로 합니다. 이 프로젝트는 CPU의 성능과 효율성을 향상시키는 데 중점을 두며, 각 단계의 설계를 통해 고성능 파이프라인 CPU를 구현하고자 합니다.

II. 구성요소 (16bit CPU제작)

1. Opcode

1. ALU_OP (4'b1111)

- Description: ALU 연산을 수행하기 위한 opcode입니다. funcCode에 따라 다양한 연산을 수행합니다.
- Function Codes:
 - 000000 (FUNC_ADD): Addition (ADD)
 - 000001 (FUNC_SUB): Subtraction (SUB)
 - 000010 (FUNC_AND): Bitwise AND (AND)
 - 000011 (FUNC_ORR): Bitwise OR (ORR)
 - 000100 (FUNC_NOT): Bitwise NOT (NOT)
 - 000101 (FUNC_TCP): Two's Complement (TCP)
 - 000110 (FUNC_SHL): Logical Shift Left (SHL)
 - 000111 (FUNC_SHR): Logical Shift Right (SHR)
 - 011011 (FUNC_RWD): Read Word Data (RWD)
 - 011100 (FUNC_WWD): Write Word Data (WWD)
 - 011001 (FUNC_JPR): Jump to Register (JPR)
 - 011010 (FUNC_JRL): Jump and Link to Register (JRL)

2. ADI_OP (4'b1000)

- Description: Immediate 값을 레지스터에 더하는 명령어입니다.
- Format:

```
| 15-12 | 11-10 | 9-8 | 7-0 |  
| opcode| rs1  | rd  | imm |
```

3. ORI_OP (4'b1010)

- Description: Immediate 값과 레지스터 값 간의 비트 OR 연산을 수행하는 명령어입니다.
- Format:

```
| 15-12 | 11-10 | 9-8 | 7-0 |  
| opcode| rs1  | rd  | imm |
```

4. LHI_OP (4'b1100)

- Description: Immediate 값을 레지스터의 상위 비트에 로드하는 명령어입니다.
- Format:

```
| 15-12 | 11-10 | 9-8 | 7-0 |  
| opcode| rs1  | rd  | imm |
```

5. LWD_OP (4'b0111)

- Description: 메모리에서 레지스터로 단어를 로드하는 명령어입니다.
- Format:

```
| 15-12 | 11-10 | 9-8 | 7-0 |  
| opcode| rs1  | rd  | imm |
```

6. SWD_OP (4'b1000)

- Description: 레지스터의 단어를 메모리에 저장하는 명령어입니다.
- Format:

```
| 15-12 | 11-10 | 9-8 | 7-0 |  
| opcode| rs1  | rd  | imm |
```

7. BNE_OP (4'b0000)

- Description: 두 레지스터의 값이 다를 경우 지정된 주소로 분기하는 명령어입니다.
- Format:

```
| 15-12 | 11-10 | 9-8 | 7-0 |  
| opcode| rs1  | rs2 | addr|
```

8. BEQ_OP (4'b0001)

- Description: 두 레지스터의 값이 같을 경우 지정된 주소로 분기하는 명령어입니다.
- Format:

```
| 15-12 | 11-10 | 9-8 | 7-0 |  
| opcode| rs1  | rs2 | addr|
```

9. BGZ_OP (4'b0010)

- Description: 레지스터의 값이 0보다 클 경우 지정된 주소로 분기하는 명령어입니다.
- Format:

15-12 11-10 9-8 7-0
opcode rs1 rd addr

10. BLZ_OP (4'b0011)

- **Description:** 레지스터의 값이 0보다 작을 경우 지정된 주소로 분기하는 명령어입니다.
- **Format:**

15-12 11-10 9-8 7-0
opcode rs1 rd addr

11. JMP_OP (4'b1101)

- **Description:** 지정된 주소로 무조건 분기하는 명령어입니다.
- **Format:**

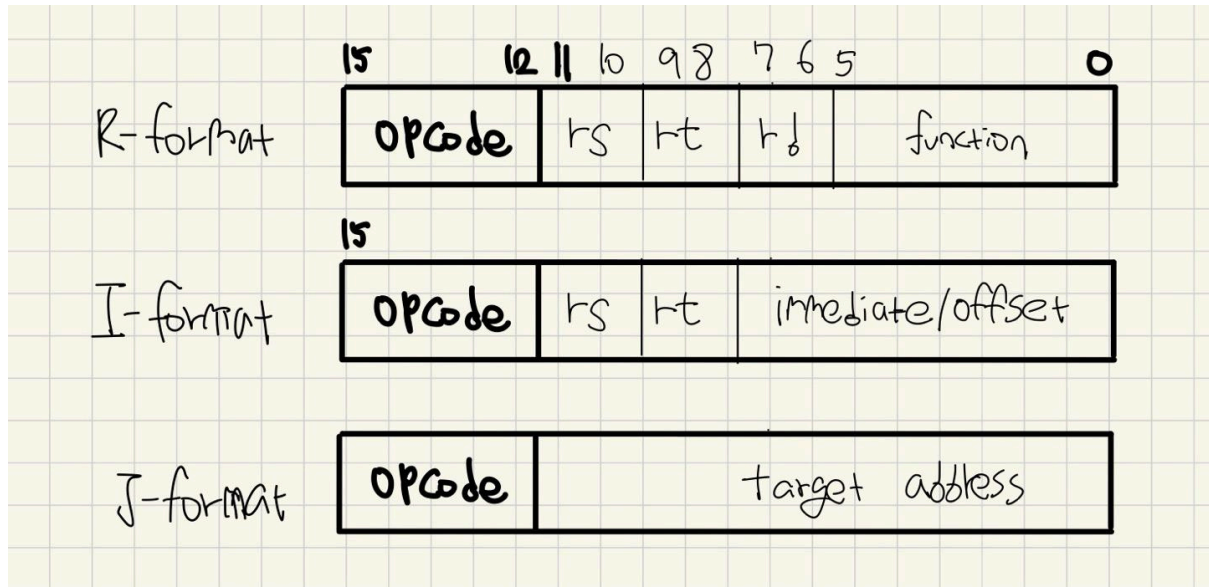
15-12 11-0
opcode addr

12. JAL_OP (4'b1110)

- **Description:** 지정된 주소로 무조건 분기하고, 현재 PC 값을 링크 레지스터에 저장하는 명령어입니다.
- **Format:**

5-12 11-0
opcode addr

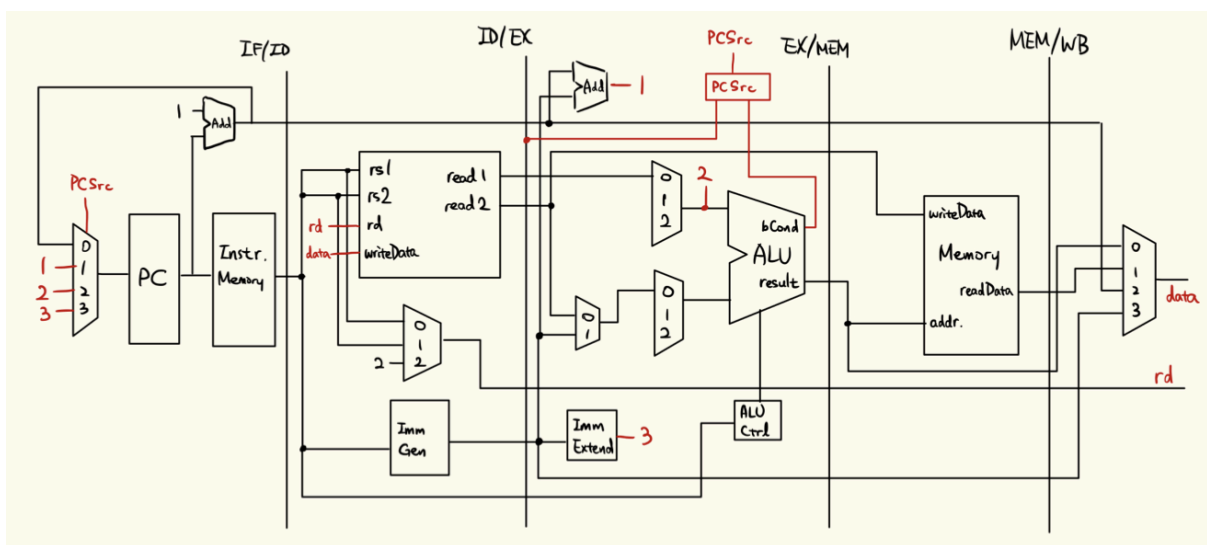
2. Mini-CPU



- R-format (Opcode 1)
레지스터 간의 연산을 수행합니다.
- I-format (Opcode 2,3,4,5,6,7,8,9,10)
immediate 값을 사용하여 동작합니다.
- J-format (Opcode 11,12)
jump와 관련된 명령어로, 점프 후의 주소를 지정합니다.
- 16비트 중 [15:12]에 해당하는 부분은 opcode로써, R-type의 경우에는 '1111'의 값을 가지고, I-type 과 J-type의 경우에는 각각 명령어 별로 고유한 값을 가지게 됩니다.
- 위 과정을 통해 R-type 여부를 확인하고, 만약 R-type 이라면 [5:0]의 값을 이용하여 각각의 명령어를 구분합니다.

III. 회로도 + 파이프라인

1. 회로도



각 모듈의 연결

Instruction Fetch

- 입력: clk, reset, pc
- 출력: instruction

Instruction Decode

- 입력: instruction
- 출력: opcode, src_addr1, src_addr2, dest_addr, operand

Execute

- 입력: opcode, operand, regA, regB
- 출력: result, N, Z

RegisterFile-Write Back (WB) 단계

- 입력: clk, reset, write_enable, read_addr1, read_addr2, write_addr, write_data

- 출력: regA, regB

ALU

- 입력: A, B, opcode
- 출력: result, N, Z

2. 파이프라인

Clock	1	2	3	4	5	6	7	8	9		
Inst1	IF	ID	EX	MEM	WB						
Inst2		IF	ID	EX	MEM	WB					
Inst3			IF	ID	EX	MEM	WB				
Inst4				IF	ID	EX	MEM	WB			
Inst5					IF	ID	EX	MEM	WB		
Inst6						IF	ID	EX	MEM	WB	
Inst7							IF	ID	EX	MEM	WB

1. Fetch (명령어 인출) 단계: 메모리에서 명령어를 가져옵니다.

2. Decode (명령어 해독) 단계: 명령어를 해독하여 필요한 제어 신호를 생성하고, 레지스터 파일에서 피연산자를 읽어옵니다.

3. Execute (명령어 실행) 단계: ALU 를 사용하여 연산을 수행합니다.

4. Memory Access (메모리 접근) 단계: 메모리 읽기 또는 쓰기 작업을 수행합니다.

5. Write Back (결과 기록) 단계: 연산 결과를 레지스터 파일에 저장합니다.

3. HAZARD 해결 방안

HazardUnit 모듈과 ForwardUnit 모듈을 사용해서 해저드 문제를 해결하려고 합니다.

- HazardUnit 모듈:
 - 파이프라인 스테이지 간의 해저드 처리
 - 입력: 파이프라인 스테이지 간의 제어 및 신호
 - 출력: 파이프라인 동작을 제어하는 제어 신호
 - 초기화 과정: 출력 레지스터 초기화 및 해저드 처리 논리 설정
 - 해저드 처리: 메모리 읽기 활성화 및 목적 레지스터 충돌 시
- ForwardUnit 모듈:
 - 파이프라인 스테이지 간의 데이터 **forwarding** 수행
 - 입력: 파이프라인 스테이지 간의 데이터 및 제어 신호
 - 출력: 데이터가 전달되는 방향을 나타내는 제어 신호
 - 초기화 과정: 출력 레지스터 초기화 및 데이터 **forwarding** 처리 논리 설정
 - 데이터 **forwarding**: 현재 스테이지에서 사용하는 레지스터와 메모리 또는 레지스터에서 읽기가 활성화될 때

IV. 코드

1. 파이프라인별 모듈

1. Fetch 단계

- 즉시 생성기 (ImmediateGenerator): 명령어에 기반하여 즉시 값을 생성합니다.
- PC 모듈: 프로그램 카운터를 관리합니다.
- 다중화기 모듈 (MUX2to1, MUX4to1, MUXPC): 데이터 입력을 선택하기 위해 사용됩니다.
- IFControl: 명령어 페치 스테이지를 제어합니다.
- GeneralPipeline: 명령어 인출을 위한 파이프라인 스테이지입니다.

2. Decode 단계

- 즉시 확장기 (ImmExtender): 산술 연산을 위해 즉시 값을 확장합니다.
- 다중화기 모듈(MUX2to1, MUX4to1, MUXPC)): 데이터 입력을 선택하기 위해 사용됩니다.
- IDPipeline: 명령어 해석을 위한 파이프라인 스테이지입니다.

3. Execute 단계

- ALUControl: ALU 작업을 결정합니다.
- EXControl: 명령어 실행 스테이지를 제어합니다.
- EXPipeline: 명령어 실행을 위한 파이프라인 스테이지입니다.

4. Memory 단계

- MEMControl: 메모리 접근 스테이지를 제어합니다.
- MEMPipeline: 메모리 접근을 위한 파이프라인 스테이지입니다.

5. Write-back 단계

- WBControl: 쓰기 백 스테이지를 제어합니다.
- WBPipeline: 쓰기 백을 위한 파이프라인 스테이지입니다.

CPU 모듈은 모든 파이프라인 스테이지를 통합하는 최상위 모듈입니다.

2. 각 단계별 코드 설명

2-1. Fetch단계

```
module GeneralPipeline(clk, inPC, inNumInst, inOpCode, inFuncCode, inRS1, inRS2, inRD, inWriteFlag,
                      PC, numInst, opCode, funcCode, rs1, rs2, rd, writeFlag);

    input clk;
    input [3:0] inOpCode;
    input [5:0] inFuncCode;
    input [1:0] inRS1;
    input [1:0] inRS2;
    input [2:0] inRD;
    input inWriteFlag;
    input [`WORD_SIZE-1:0] inPC;
    input inNumInst;

    output [3:0] opCode;
    output [5:0] funcCode;
    output [1:0] rs1;
    output [1:0] rs2;
    output [2:0] rd;
    output [`WORD_SIZE-1:0] PC;
    output numInst;
    output writeFlag;

    reg [3:0] opCode;
    reg [5:0] funcCode;
    reg [1:0] rs1;
    reg [1:0] rs2;
    reg [2:0] rd;
    reg [`WORD_SIZE-1:0] PC;
    reg numInst;
    reg writeFlag;

    initial begin
        opCode = 0;
        funcCode = 0;
        rs1 = 0;
        rs2 = 0;
        rd = 4;
        PC = 0;
        numInst = 0;
    end

    always @(posedge clk) begin
        writeFlag = inWriteFlag;
        opCode = inOpCode;
        funcCode = inFuncCode;
        rs1 = inRS1;
        rs2 = inRS2;
        rd = inRD;
        PC = inPC;
        numInst = inNumInst;
    end

endmodule
```

이 코드는 실행(Execute) 파이프라인 스테이지를 구성하는 모듈입니다. 클럭 신호와 여러 입력을 받아들이고, 해당 스테이지에서 필요한 제어 신호 및 출력을 생성합니다.

클럭의 상승 에지에 따라 입력으로 받은 명령어와 데이터를 업데이트하고, 제어 플래그에 따라 명령어의 수행 여부를 결정합니다. 수행할 때마다 명령어와 데이터를 새롭게 가져오고, 실행에 필요한 제어 신호를 생성합니다.

여기서 생성된 제어 신호는 메모리 읽기 신호(memRead), ALU 소스(ALUSrc), 프로그램 카운터 제어(PCCtrl), 그리고 쓰기 플래그(writeFlag) 등이 있습니다. 이러한 제어 신호는 해당 파이프라인 스테이지에서 명령어를 올바르게 수행하고 다음 단계로 데이터를 전달하는 데 사용됩니다.

2-2. Decode 단계

```
module IDPipeline(clk, inInstruction, inPC, instruction, PC, rdSelector, writeFlag, flushFlag);
    input clk;
    input [`WORD_SIZE-1:0] inInstruction;
    input [`WORD_SIZE-1:0] inPC;
    input writeFlag;
    input flushFlag;

    output [`WORD_SIZE-1:0] instruction;
    output [`WORD_SIZE-1:0] PC;
    output [1:0] rdSelector;

    reg [`WORD_SIZE-1:0] instruction;
    reg [`WORD_SIZE-1:0] PC;
    reg [1:0] rdSelector;

    reg [3:0] opCode;
    reg [5:0] funcCode;

    initial begin
        instruction = 0;
        PC = 0;
        rdSelector = 3;
    end

    always @(posedge clk) begin
        if(writeFlag) begin
            instruction = inInstruction;
            opCode = instruction[15:12];
            funcCode = instruction[5:0];
            PC = inPC;

            if((opCode == `ALU_OP && funcCode != `INST_FUNC_JPR && funcCode != 6'd29 && funcCode != 6'd28) || opCode == `JAL_OP || opCode == `LWD_OP || opCode == `ADI_OP || opCode == `ORI_OP ||
                (opCode == `JAL_OP || (opCode == `JRL_OP && funcCode == `INST_FUNC_JRL)) ? 2 :
                (opCode != `ALU_OP && opCode != `SND_OP ? 1 : 0);
            else
                rdSelector = 3;
            end

        if(flushFlag) begin
            instruction = 0;
            PC = 0;
            rdSelector = 3;
        end
    end
endmodule
```

이 코드는 ID 파이프라인 스테이지의 제어를 담당하는 모듈입니다. 클럭 신호 및 입력 명령어, 프로그램 카운터, 쓰기 플래그, 그리고 플러시 플래그를 입력으로 받아들이며, 현재 명령어, 프로그램 카운터, 그리고 레지스터 선택자를 출력합니다.

클럭 신호가 상승 엣지일 때, 입력 명령어와 프로그램 카운터를 업데이트하고, 명령어에서 **opcode**와 **funcCode**를 추출하여 레지스터에 저장합니다.

현재 명령어가 **ALU** 작업이거나 점프, 데이터 로드/저장과 관련된 명령어일 때, 적절한 레지스터를 선택하여 레지스터 선택자를 설정합니다. 그렇지 않으면 레지스터 쓰기를

비활성화합니다.

플러시 플래그가 활성화되면 모든 출력을 초기화하여 파이프라인을 비웁니다.

2-3. Execute 단계

```
module EXPipeline(clk, inPC, inNumInst, inOpCode, inFuncCode, inRS1, inRS2, inRD, inImm, inReadData1, inReadData2, inWriteFlag,
                 memRead, ALUSrc, PC, numInst, opCode, funcCode, rs1, rs2, rd, imm, readData1, readData2, PCCtrl, writeFlag);

    input clk;
    input [3:0] inOpCode;
    input [5:0] inFuncCode;
    input [1:0] inRS1;
    input [1:0] inRS2;
    input [2:0] inRD;
    input [WORD_SIZE-1:0] inImm;
    input [WORD_SIZE-1:0] inReadData1, inReadData2;
    input [WORD_SIZE-1:0] inPC;
    input inNumInst;
    input inWriteFlag;

    output memRead;
    output ALUSrc;
    output [3:0] opCode;
    output [5:0] funcCode;
    output [1:0] rs1;
    output [1:0] rs2;
    output [2:0] rd;
    output [WORD_SIZE-1:0] imm;
    output [WORD_SIZE-1:0] readData1, readData2;
    output [WORD_SIZE-1:0] PC;
    output numInst;
    output [1:0] PCCtrl;
    output writeFlag;

    reg [WORD_SIZE-1:0] imm;
    reg [WORD_SIZE-1:0] readData1, readData2;
    reg [1:0] PCCtrl;

    GeneralPipeline EXGeneralPipeline(clk, inPC, inNumInst, inOpCode, inFuncCode, inRS1, inRS2, inRD, inWriteFlag, PC, numInst, opCode, funcCode, rs1, rs2, rd, writeFlag);
    EXControl Control(opCode, funcCode, memRead, ALUSrc, writeFlag);

    initial begin
        imm = 0;
        readData1 = 0;
        readData2 = 0;
        PCCtrl = 0;
    end

    always @(posedge clk) begin
        imm = inImm;
        readData1 = inReadData1;
        readData2 = inReadData2;
        if(writeFlag == 1) begin
            PCCtrl =
                (inOpCode == `BNE_OP || inOpCode == `BEQ_OP || inOpCode == `BGZ_OP || inOpCode == `BLZ_OP) ? 2'b01 :
                (inOpCode == `JMP_OP || inOpCode == `JAL_OP) ? 2'b10 :
                (
                    inOpCode == `JPR_OP && inFuncCode == `INST_FUNC_JPR ||
                    inOpCode == `JRL_OP && inFuncCode == `INST_FUNC_JRL
                ) ? 2'b11 : 2'b00;
        end
        else begin
            PCCtrl = 0;
        end
    end
end
endmodule
```

이 코드는 실행(Execute) 파이프라인 스테이지를 구성하는 모듈입니다. 클럭 신호와 여러 입력을 받아들이고, 해당 스테이지에서 필요한 제어 신호 및 출력을 생성합니다.

클럭의 상승 에지에 따라 입력으로 받은 명령어와 데이터를 업데이트하고, 제어 플래그에 따라 명령어의 수행 여부를 결정합니다. 수행할 때마다 명령어와 데이터를 새롭게 가져오고, 실행에 필요한 제어 신호를 생성합니다.

여기서 생성된 제어 신호는 메모리 읽기 신호(memRead), ALU 소스(ALUSrc), 프로그램

카운터 제어(PCCtrl), 그리고 쓰기 플래그(writeFlag) 등이 있습니다. 이러한 제어 신호는 해당 파이프라인 스테이지에서 명령어를 올바르게 수행하고 다음 단계로 데이터를 전달하는 데 사용됩니다.

2-4. Memory 단계

```

512 module MEMPipeline(clk, inPC, inNumInst, inOpCode, inFuncCode, inRS1, inRS2, inRD, inImmExtend, inALUResult, inReadData1, inReadData2, inMemRead, inWriteFlag,
513                      memRead, memWrite, regWrite, PC, numInst, opCode, funcCode, rs1, rs2, rd, immExtend, ALUResult, readData1, readData2, writeFlag);
514     input clk;
515     input [3:0] inOpCode;
516     input [5:0] inFuncCode;
517     input [1:0] inRS1;
518     input [1:0] inRS2;
519     input [2:0] inRD;
520     input [`WORD_SIZE-1:0] inImmExtend;
521     input [`WORD_SIZE-1:0] inALUResult;
522     input [`WORD_SIZE-1:0] inPC;
523     input inNumInst;
524     input [`WORD_SIZE-1:0] inReadData1;
525     input [`WORD_SIZE-1:0] inReadData2;
526     input inMemRead;
527     input inWriteFlag;
528
529     output memRead;
530     output memWrite;
531     output regWrite;
532     output [3:0] opCode;
533     output [5:0] funcCode;
534     output [1:0] rs1;
535     output [1:0] rs2;
536     output [2:0] rd;
537     output [`WORD_SIZE-1:0] immExtend;
538     output [`WORD_SIZE-1:0] ALUResult;
539     output [`WORD_SIZE-1:0] PC;
540     output numInst;
541     output [`WORD_SIZE-1:0] readData1;
542     output [`WORD_SIZE-1:0] readData2;
543     output writeFlag;
544
545     reg memRead;
546     reg [`WORD_SIZE-1:0] immExtend;
547     reg [`WORD_SIZE-1:0] ALUResult;
548     reg [`WORD_SIZE-1:0] readData1;
549     reg [`WORD_SIZE-1:0] readData2;
550
551     GeneralPipeline MEMGeneralPipeline(clk, inPC, inNumInst, inOpCode, inFuncCode, inRS1, inRS2, inRD, inWriteFlag, PC, numInst, opCode, funcCode, rs1, rs2, rd, writeFlag);
552     MEMControl Control(opCode, funcCode, memWrite, regWrite, writeFlag);
553
554     initial begin
555         immExtend = 0;
556         ALUResult = 0;
557         readData1 = 0;
558         readData2 = 0;
559         memRead = 0;
560     end
561
562     always @(posedge clk) begin
563         immExtend = inImmExtend;
564         ALUResult = inALUResult;
565         readData1 = inReadData1;
566         readData2 = inReadData2;
567
568         if(writeFlag == 1) begin
569             memRead = inMemRead;
570         end
571         else begin
572             memRead = 0;
573         end
574     end
575 endmodule

```

이 모듈은 메모리 액세스 스테이지를 구현합니다. 이 스테이지에서는 주로 메모리에서 데이터를 읽거나 쓰는 작업을 수행합니다. 입력 신호들(inOpCode, inFuncCode, inRS1, inRS2, inRD, inImmExtend, inALUResult, inReadData1, inReadData2, inMemRead, inWriteFlag)은 이전 스테이지에서 전달된 정보를 나타냅니다.

출력 신호들(opCode, funcCode, rs1, rs2, rd, immExtend, ALUResult, readData1, readData2,

memRead, memWrite, regWrite, PC, numInst, writeFlag)은 현재 스테이지에서 사용될 정보를 나타냅니다. 이 모듈은 입력 값을 레지스터에 저장한 후 클럭의 상승 에지에서 해당하는 레지스터 값을 출력으로 내보냅니다. 클럭이 상승할 때마다 입력 값을 레지스터에 복사하고, 쓰기 플래그가 활성화되어 있는 경우에만 메모리에서 데이터를 읽습니다.

2-5. Write-back 단계

```

577 module WBPipeline(clk, inPC, inNumInst, inOpCode, inFuncCode, inRS1, inRS2, inRD, inImmExtend, inALUResult, inReadData, inReadData1, inWriteFlag,
578                   dataSelector, regWrite, isHalt, isWWD, PC, numInst, opCode, funcCode, rs1, rs2, rd, immExtend, ALUResult, readData, readData1, writeFlag);
579     input clk;
580     input [3:0] inOpCode;
581     input [5:0] inFuncCode;
582     input [1:0] inRS1;
583     input [1:0] inRS2;
584     input [2:0] inRD;
585     input [WORD_SIZE-1:0] inImmExtend;
586     input [WORD_SIZE-1:0] inALUResult;
587     input [WORD_SIZE-1:0] inPC;
588     input inNumInst;
589     input [WORD_SIZE-1:0] inReadData;
590     input [WORD_SIZE-1:0] inReadData1;
591     input inWriteFlag;
592
593     output [1:0] dataSelector;
594     output regWrite;
595     output isHalt;
596     output isWWD;
597     output [3:0] opCode;
598     output [5:0] funcCode;
599     output [1:0] rs1;
600     output [1:0] rs2;
601     output [2:0] rd;
602     output [WORD_SIZE-1:0] immExtend;
603     output [WORD_SIZE-1:0] ALUResult;
604     output [WORD_SIZE-1:0] PC;
605     output numInst;
606     output [WORD_SIZE-1:0] readData;
607     output [WORD_SIZE-1:0] readData1;
608     output writeFlag;
609
610     reg [WORD_SIZE-1:0] immExtend;
611     reg [WORD_SIZE-1:0] readData;
612     reg [WORD_SIZE-1:0] readData1;
613     reg [WORD_SIZE-1:0] ALUResult;
614
615     GeneralPipeline WBGeneralPipeline(clk, inPC, inNumInst, inOpCode, inFuncCode, inRS1, inRS2, inRD, inWriteFlag, PC, numInst, opCode, funcCode, rs1, rs2, rd, writeFlag);
616     WBControl Control(opCode, funcCode, dataSelector, regWrite, isHalt, isWWD, writeFlag);
617
618     initial begin
619         immExtend = 0;
620         ALUResult = 0;
621         readData = 0;
622         readData1 = 0;
623     end
624
625     always @(posedge clk) begin
626         immExtend = inImmExtend;
627         ALUResult = inALUResult;
628         readData = inReadData;
629         readData1 = inReadData1;
630     end
631 endmodule

```

이 모듈은 쓰기 백 스테이지를 구현합니다. 이 스테이지에서는 실행 스테이지에서 계산된 결과를 레지스터 파일에 쓰거나, 프로그램 카운터(PC)를 업데이트하거나, 기타 작업을 수행합니다. 입력 신호들(inOpCode, inFuncCode, inRS1, inRS2, inRD, inImmExtend, inALUResult, inReadData, inReadData1, inWriteFlag)은 이전 스테이지에서 전달된 정보를 나타냅니다. 출력 신호들(opCode, funcCode, rs1, rs2, rd, immExtend, ALUResult, readData, readData1, dataSelector, regWrite, isHalt, isWWD, PC, numInst, writeFlag)은 현재 스테이지에서 사용될 정보를 나타냅니다.

이 모듈은 입력 값을 레지스터에 저장한 후 클럭의 상승 에지에서 해당하는 레지스터 값을 출력으로 내보냅니다. 클럭이 상승할 때마다 입력 값을 레지스터에 복사하고, 출력 값을 갱신합니다. 초기화 부분에서는 출력 레지스터들의 초기값을 설정합니다. 출력 레지스터들은 클럭의 상승 에지에 의해 입력 값으로 갱신됩니다. 따라서 해당 모듈은

파이프라인 CPU에서 쓰기 백 스테이지를 제어하는 역할을 합니다.

2-6. CPU

```
732 module cpu(clk, reset_n, readM1, address1, data1, readM2, writeM2, address2, data2, num_inst, output_port, is_halted);
733     input clk;
734     wire clk;
735     input reset_n;
736     wire reset_n;
737
738     output readM1;
739     wire readM1;
740     output [`WORD_SIZE-1:0] address1;
741     wire [`WORD_SIZE-1:0] address1;
742     output readM2;
743     wire readM2;
744     output writeM2;
745     wire writeM2;
746     output [`WORD_SIZE-1:0] address2;
747     wire [`WORD_SIZE-1:0] address2;
748
749     input [`WORD_SIZE-1:0] data1;
750     wire [`WORD_SIZE-1:0] data1;
751     inout [`WORD_SIZE-1:0] data2;
752     wire [`WORD_SIZE-1:0] data2;
753
754     output [`WORD_SIZE-1:0] num_inst;
755     reg [`WORD_SIZE-1:0] num_inst;
756     output [`WORD_SIZE-1:0] output_port;
757     reg [`WORD_SIZE-1:0] output_port;
758     output is_halted;
759     wire is_halted;
760     // TODO : Implement your pipelined CPU!
761
762     wire [`WORD_SIZE-1:0] IF_writePC, IF_outputPC, IF_instruction, IF_PC;
763     wire [`WORD_SIZE-1:0] ID_instruction, ID_PC, ID_immediate, ID_readData1, ID_readData2;
764     wire [`WORD_SIZE-1:0] EX_PC, EX_readData1, EX_readData2, EX_immediate, EX_immExtend, EX_ALUSrc2, EX_ALUResult, EX_ALUInput1, EX_ALUInput2;
765     wire [`WORD_SIZE-1:0] MEM_PC, MEM_immExtend, MEM_immediate, MEM_ALUResult, MEM_readData, MEM_readData1, MEM_readData2;
766     wire [`WORD_SIZE-1:0] WB_PC, WB_immExtend, WB_immediate, WB_writeData, WB_ALUResult, WB_readData, WB_readData1;
```

```

768     wire IF_PCWrite;
769     wire ID_IDWrite, ID_FlushFlag, ID_numInst, ID_EXWrite;
770     wire EX_WriteFlag, EX_ALUSrc, EX_bCond, EX_numInst, EX_memRead;
771     wire MEM_memRead, MEM_memWrite, MEM_regWrite, MEM_numInst, MEM_WriteFlag;
772     wire WB_regWrite, WB_isHalt, WB_isWd, WB_numInst, WB_WriteFlag;
773
774     wire [1:0] ID_rs1, ID_rs2, ID_rdSelector;
775     wire [1:0] EX_rs1, EX_rs2, EX_forwardA, EX_forwardB, EX_PCctrl;
776     wire [1:0] MEM_rs1, MEM_rs2;
777     wire [1:0] WB_rs1, WB_rs2, WB_dataSelector;
778     wire [1:0] PCctrl;
779
780     wire [2:0] ID_rd, EX_rd, MEM_rd, WB_rd;
781
782     wire [3:0] ID_opCode, EX_opCode, EX_ALUOp, MEM_opCode, WB_opCode;
783     wire [5:0] ID_funcCode, EX_funcCode, MEM_funcCode, WB_funcCode;
784
785     assign readM1 = reset_n;
786     assign readM2 = MEM_memRead;
787     assign writeM2 = MEM_memWrite;
788     assign address1 = IF_outputPC;
789     assign address2 = MEM_ALUResult;
790     assign data2 = MEM_memWrite ? MEM_readData2 : `WORD_SIZE'bz;
791     assign IF_instruction = data1;
792     assign MEM_readData = data2;
793     assign is_halted = WB_isHalt;
794     assign PCctrl = (EX_bCond ? EX_PCctrl :
795                     (EX_PCctrl == 2'b01 ? 2'b00 : EX_PCctrl));
796
797     // Extra
798     HazardUnit hazardUnit(reset_n, ID_rs1, ID_rs2, EX_rd, EX_memRead, IF_PCWrite, ID_IDWrite, ID_FlushFlag, ID_EXWrite, PCctrl);
799     ForwardUnit forwardUnit(EX_rs1, EX_rs2, MEM_rd, WB_rd, EX_opCode, EX_funcCode, MEM_regWrite, WB_regWrite, EX_forwardA, EX_forwardB);
800
801     // IF stage
802     MUX4to1 PCSelector(IF_outputPC + 1'd1, EX_PC + 1'd1 + EX_immediate, EX_immExtend, EX_ALUInput1, PCctrl, IF_writePC);
803     PC pc(IF_writePC, IF_outputPC, clk, reset_n, IF_PCWrite);
804     // Instruction memory access
805
806     // ID stage
807     assign ID_rs1 = ID_instruction[11:10];
808     assign ID_rs2 = ID_instruction[9:8];
809     assign ID_opCode = ID_instruction[15:12];
810     assign ID_funcCode = ID_instruction[5:0];
811     assign ID_numInst = (ID_EXWrite && reset_n);
812     IDPipeline Pipeline_ID(clk, IF_instruction, IF_outputPC, ID_instruction, ID_PC, ID_rdSelector, ID_IDWrite, ID_FlushFlag);
813     MUX8PC MUX_ID_rd(ID_instruction[7:6] + 3'b000, ID_instruction[9:8] + 3'b000, 3'b10, ID_rdSelector, ID_rd);
814
815     Register register(clk, ID_rs1, ID_rs2, WB_rd, WB_writeData, ID_readData1, ID_readData2, WB_regWrite, reset_n);
816     ImmediateGenerator immGen(ID_instruction, ID_immediate);
817
818     // EX stage
819     EXPipeline Pipeline_EX(clk, ID_PC, ID_numInst, ID_opCode, ID_funcCode, ID_rs1, ID_rs2, ID_rd, ID_immediate, ID_readData1, ID_readData2, (ID_EXWrite && reset_n),
820                           EX_memRead, EX_ALUSrc, EX_PC, EX_numInst, EX_opCode, EX_funcCode, EX_rs1, EX_rs2, EX_rd, EX_immediate, EX_readData1, EX_PCctrl, EX_WriteFlag);
821     ImmExtender immExtender(EX_PC, EX_immediate, EX_immExtend);
822     MUX4to1 MUX_EX_ALUSrc2(EX_readData2, EX_immediate, EX_ALUSrc, EX_ALUSrc2);
823     MUX4to1 MUX_EX_ALUInput1(EX_readData1, WB_writeData, MEM_ALUResult, `WORD_SIZE'b0, EX_forwardA, EX_ALUInput1);
824     MUX4to1 MUX_EX_ALUInput2(EX_ALUSrc2, MEM_ALUResult, WB_writeData, `WORD_SIZE'b0, EX_forwardB, EX_ALUInput2);
825     ALUControl aluControl(EX_opCode, EX_funcCode, EX_ALUOp);
826     ALU alu(EX_ALUInput1, EX_ALUInput2, EX_ALUOp, EX_ALUResult, EX_bCond);
827
828     // MEM stage
829     MEMPipeline Pipeline_MEM(clk, EX_PC, (PCctrl != 0 ? 1'b0 : EX_numInst), EX_opCode, EX_funcCode, EX_rs1, EX_rs2, EX_rd, EX_immediate, EX_ALUResult, EX_ALUInput1, EX_readData1, EX_memRead, EX_WriteFlag,
830                             MEM_memRead, MEM_memWrite, MEM_regWrite, MEM_PC, MEM_numInst, MEM_opCode, MEM_funcCode, MEM_rs1, MEM_rs2, MEM_rd, MEM_immediate, MEM_ALUResult, MEM_readData1, MEM_readData2, MEM
831
832     // WB stage
833     WBPipeline Pipeline_WB(clk, MEM_PC, MEM_numInst, MEM_opCode, MEM_funcCode, MEM_rs1, MEM_rs2, MEM_rd, MEM_immediate, MEM_ALUResult, MEM_readData1, MEM_readData2, MEM_WriteFlag,
834                           WB_dataSelector, WB_regWrite, WB_isHalt, WB_isWd, WB_PC, WB_numInst, WB_opCode, WB_funcCode, WB_rs1, WB_rs2, WB_rd, WB_immediate, WB_ALUResult, WB_readData1, WB_readData2, WB_Wr
835     MUX4to1 MUX_WB_writeData(WB_ALUResult, WB_readData, WB_PC + `WORD_SIZE'b1, WB_immediate, WB_dataSelector, WB_writeData);

```

이 CPU 모듈은 파이프라인 아키텍처를 가진 중앙 처리 장치를 나타냅니다. 클럭 및 리셋 신호를 입력으로 받아, 여러 단계로 구성된 데이터 및 제어 신호를 처리합니다. 입력 및 출력: 클럭 및 리셋 신호 외에도 다양한 데이터 및 제어 신호를 입력으로 받고, 명령어 실행 결과 및 상태를 출력으로 내보냅니다.

파이프라인 구조: 명령어 처리를 위해 여러 단계로 구성된 파이프라인을 사용합니다. 각 단계에서는 입력 데이터를 처리하고 결과를 다음 단계로 전달합니다.

주요 단계

1. 명령어 가져오기(IF): 프로그램 카운터 값을 사용하여 명령어를 메모리에서 가져옵니다.

2. 명령어 해석(ID): 가져온 명령어를 해석하고 실행에 필요한 제어 신호를 생성합니다.
3. 명령어 실행(EX): 명령어를 실행하고 연산을 수행합니다.
4. 메모리 접근(MEM): 데이터 메모리에 접근하여 데이터를 읽거나 쓰기를 수행합니다.
5. 결과 쓰기(WB): 실행 결과를 레지스터 파일에 쓰거나 필요한 출력을 생성합니다.

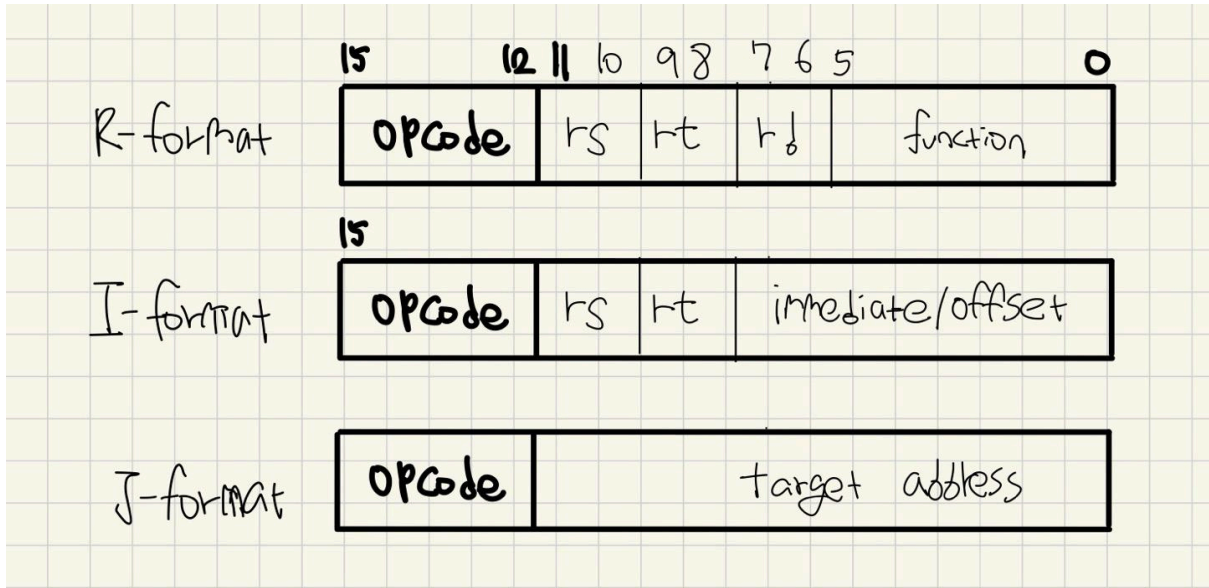
제어 및 레지스터: 각 단계에서는 해당하는 제어 장치와 레지스터를 사용하여 데이터 및 제어 신호를 관리합니다.

추가 기능: 위험 상황을 처리하기 위한 **Hazard Unit** 및 데이터 전달을 위한 **Forward Unit**이 구현되어 있습니다.

상태 관리 및 출력: 리셋 시 초기화되며, 실행된 명령어의 수 및 출력 데이터를 관리하고 필요한 출력을 생성합니다.

이 **CPU** 모듈은 파이프라인 아키텍처를 갖춘 **CPU**를 간단하게 표현하고, 다양한 명령어의 실행을 관리하는 데 사용됩니다.

V. 수행 결과



Simulation 실행

TestID[4] <= "2-1"; TestAns[4] <= 16'h0001; TestPassed[4] <= 1'bx;

위와 같은 방식으로 시뮬레이션 이전 예상되는 값을 미리 입력하고 명령어 실행결과가 예상값과 같다면 TestPassed[] 값이 1이 되도록 설정했습니다.

Case	Instruction	Opcode	Function	Format	Ans
2-1	ADI	0100	-	I	0001
3-1	ORI	0101	-	I	0001
4-1	ADD	1111	0	R	0010
5-1	SUB	1111	0001	R	0010
6-1	AND	1111	0010	R	0000
7-1	ORR	1111	0011	R	0010
8-1	NOT	1111	0100	R	FFFD
9-1	TCP	1111	0101	R	FFFF
10-1	SHL	1111	0110	R	0100

11-1	SHR	1111	0111	R	0001
13-1	SWD	1000	-	I	FFFF
	LWD	0111	-	I	
14-2	JMP	1001	-	J	0001
	ENI	1111	11110	R	
	WWD	1111	11100	R	
15-2	BNE	0000	-	I	0001
	ENI	1111	11110	R	
	WWD	1111	11100	R	
16-1	BEQ	0001	-	I	FFFF
	ENI	1111	11110	R	
	WWD	1111	11100	R	

2-1

memory[16'h2b] <= 16'h4401;

0100 01 00 00000001

ADI R0 R1 0X01: R1에 0x01을 더한 결과를 R0에 저장합니다.

$R1 = R0 (0) + 1 = 1$

3-1

memory[16'h31] <= 16'h5502

0101 10 01 00000001

ORI R1, R2, #01: R2와 #01을 OR 연산한 결과를 R1에 저장합니다.

$R1 = R2 || 1 = 1$

4-1

memory[16'h35] <= 16'hf2c0;

1111 00 10 11 000000

ADD R3 R1 R2: $R1+R2 = R3$

$R1 + R2 = 1 + 1 = 2$

5-1

memory[16'h3b] <= 16'hf2c1;

1111 00 10 11 000001

SUB R3 R0 R2: $R3 = R0 - R2 = 3 - 1 = 2$

6-1

memory[16'h47] <= 16'hf2c2

1111 00 10 11 000010

AND R3 R0 R2: $R3 = R0 \& R2 = 0000 \& 0010 = 0000$

7-1

memory[16'h4d] <= 16'hf2c3

1111 00 10 11 000011

ORR R3 R0 R2: $R3 = R0 \parallel R2 = 0000 \parallel 0010 = 0010$

02

8-1

memory[16'h53] <= 16'hf0c4;

1111 00 00 11 000100

NOT R3, R0: R0을 NOT 연산한 결과를 R3에 저장합니다.

$R3 = \text{NOT } 0000\ 0000\ 0000\ 0001 = 1111\ 1111\ 1111\ 1110 = \text{FFFD}$

9-1

memory[16'h59] <= 16'hf0c5;

1111 00 00 11 000101

TCP R3, R0: R0을 TCP 연산한 결과를 R3에 저장합니다.

$R3 = \text{TCP } 0000\ 0000\ 0000\ 0001 = 1111\ 1111\ 1111\ 1111 = \text{FFFF}$

10-1

memory[16'h5f] <= 16'hf0c6;

1111 00 00 11 000110

SHL R3, R1: R3에 R1 <1 한 결과를 저장합니다.

$R3 = 0000\ 0000\ 0000\ 0010 \ll 1 = 0000\ 0000\ 0000\ 0100 = 4$

memory[16'h60] <= 16'hfc1c; //WWD R3

ANS 4

11-1

memory[16'h65] <= 16'hf0c7;

1111 00 00 11 000111

SHR R3, R0: R3에 R1>1 한 결과를 저장합니다.

R3 = 0000 0000 0000 0010 > 1 = 0000 0000 0000 0001 = 1

memory[16'h66] <= 16'hfc1c; //WWD R3

13-1

memory[16'h6f] <= 16'h8901;

1000 10 01 00000001

SWD R1, 2, 1: R(2+1)에 R1 값이 저장됩니다.

R3 = FFFF

memory[16'h70] <= 16'h8802;

1000 10 00 00000010

SWD R0, 2, 2: R(2+2)에 R0 값이 저장됩니다.

R4 = 0001

memory[16'h71] <= 16'h7801;

0111 10 00 00000001

LWD R0, 2, 1: R0에 R(2+1) 값을 불러옵니다.

R0 = R3 = FFFF

14-2

memory[16'h77] <= 16'h9079

1001 000001111001

JMP 0X79: PC=79로 이동합니다.

memory[16'h78] <= 16'hf01d

1111 00 00 00 011110

ENI: Enables interrupts가 발생


```
memory[16'h79] <= 16'hf41c
1111 01 00 00 011100
```

WWD R1 PC=78을 무시하고 결과값을 정상적으로 내보냅니다.

R1 = 0001

15-2

```
memory[16'h7e] <= 16'h601;
0000 01 10 00 000001
```

BNE R2 R1 1: R2!= R1 라면 PC <= PC +2를 진행합니다.

```
memory[16'h7f] <= 16'hf01d;
1111 00 00 00 011110
```

ENI: Enables interrupts가 발생

```
memory[16'h80] <= 16'hf41c
1111 01 00 00 011100
```

WWD R1 PC=7f을 무시하고 결과값을 정상적으로 내보냅니다.

R1 = 0001

16-1

```
memory[16'h81] <= 16'h1601
0001 01 10 00 000001
```

BEQ R2, R1, 0: R2= R1 이면 PC <= PC +1를 진행합니다.

```
memory[16'h82] <= 16'h9084; //JMP 0x84
memory[16'h83] <= 16'hf01d; //ENI
memory[16'h84] <= 16'hf01c; //WWD R0
```

R0 = FFFF

Clock # 1436

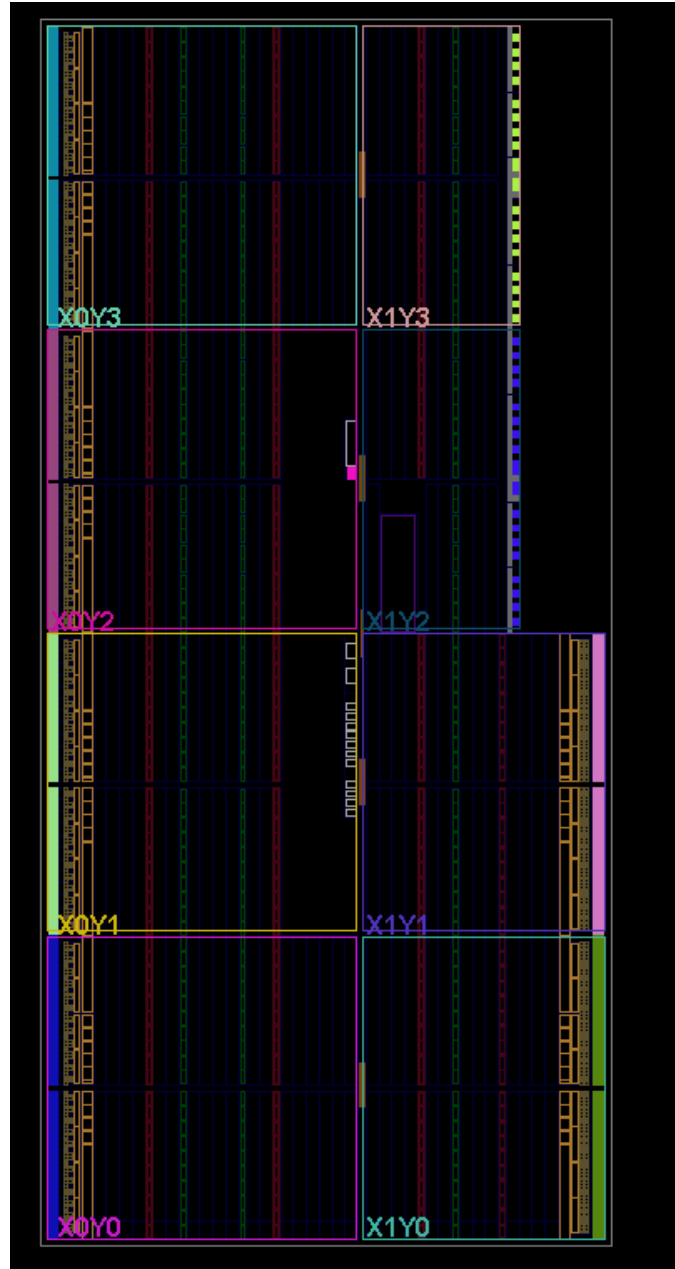
The testbench is finished. Summarizing...

All Pass!

\$finish called at time : 143750 ns : File "C:/Users/USER/Desktop/digital/pipelined-cpu-verilog-main/cpu_TB.v" Line 153

최종적으로 모든 Testbench가 예상값과 일치하는 것을 확인할 수 있습니다.

Synthesis 실행



Design Timing Summary

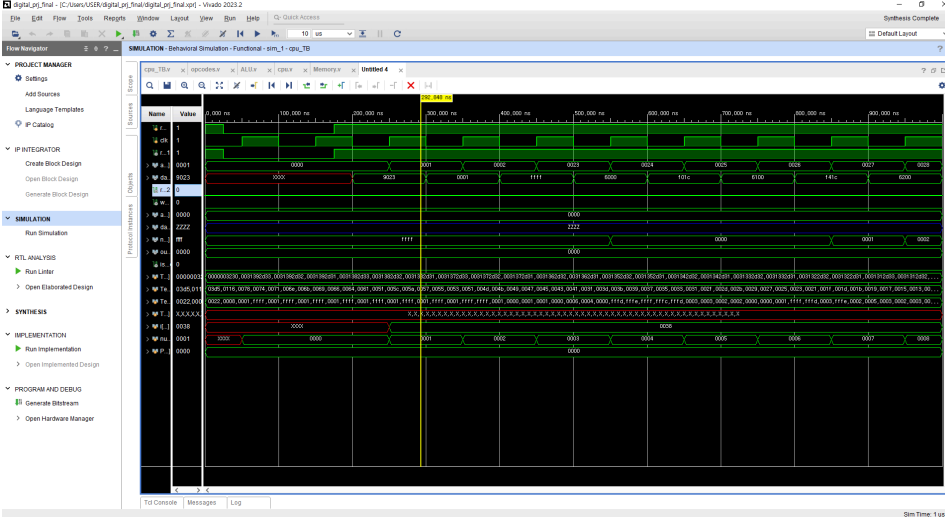
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	inf	Worst Hold Slack (WHS):	inf	Worst Pulse Width Slack (WPWS):	NA
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	NA
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	NA
Total Number of Endpoints:	603	Total Number of Endpoints:	603	Total Number of Endpoints:	NA

There are no user specified timing constraints.

VI. 프로젝트 수행일지

이름	김철중(팀장)	오예찬	이준영
모임일자	수행내용		
4/17	<ul style="list-style-type: none"> - 자기소개 및 역할분담 김철중 : 팀장 및 전체 총괄 오예찬 : 서기 및 회의록 작성 이준영 : CTO - CPU의 기본 개념 및 원리 공부 		
4/24	<ul style="list-style-type: none"> - 미니 cpu에 사용되는 연산자에 대한 공부 및 코드 구현 산술 연산자 (AND / SUB / MUL / DIV) 비교 연산자 (AND / OR / NOT) 비트 연산자 (shift(logic) / 비트 AND / OR / XOR) 		
5/1	<ul style="list-style-type: none"> - 보고서 목차 구성 - Mu0 processor 기반의 cpu 제작 방식을 모색 기능 및 구조 변경 (MUL 추가 / shift 연산 추가 / bit 수 조정) 		
5/8	<ul style="list-style-type: none"> - Execute 단계에서 사용되는 명령어 코드 작성 및 시뮬레이션 수행 LDA / STO / JGE / NOP 구현 - Fetch , Decode , Execute를 합쳐서 수행하는 과정에서 오류 발생 		
5/15	<ul style="list-style-type: none"> - Fetch , Decode , Execute를 합치는 과정에서 발생한 오류 수정 - 파이프라인 사용 여부에 대한 논의 (3 stage / 5 stage) 		
5/22	<ul style="list-style-type: none"> - 파이프라인 사용시에 발생하는 Hazard에 대한 해결 방안 모색 - imm 사용 여부에 대한 논의 - 프로젝트 코드 수정 (R / I / J -type) 		
5/29	<ul style="list-style-type: none"> - 보고서 작성 (각 단계 별 모듈 및 시뮬레이션 결과 작성) - 프로젝트 코드 최종 점검 		
6/5	<ul style="list-style-type: none"> - 발표 자료 (ppt , 대본) 제작 		

VII. Trouble shooting

<p>문제상황</p>	 <p>이전의 코드를 작성했을 때 pc 부분에서 pc=6에서 jmp가 실행되어 pc=8 성공적으로 jmp된 것을 확인할 수 있습니다. opcode, operand 또한 test에 근거해 예상되는 simulation 결과 값이 도출되었지만 result와 거기에서 파생되는 값들이 예상과 다른 값들이 출력되는 것을 확인했습니다. simulation을 통해 볼 때 fetch와 decode module에서는 오류가 발생하지 않은 것을 보이고 3 stage pipe line을 구성하면서 stage에서 시간적으로 변수에 값이 잘못 들어갔거나 flag부분을 추가하면서 값에 오류가 들어갔다고 생각하였습니다.</p>
<p>해결법</p>	<p>3stage 파이프라인을 5 stage파이프라인으로 늘려서 명령어를 좀 더 세분화하여 문제를 각 단계별로 구체적인 명령을 수행하도록 수정하였습니다.</p> <p>기존의 3단계 파이프라인에서는 명령어의 실행이 단순화되어 있을 수 있습니다.</p> <p>5단계 파이프라인에서는 명령어를 효율적으로 처리하기 위해 추가적인 단계를 도입할 수 있습니다. 플래그를 사용하면 복잡한 제어 로직을 구현해야 하지만, 플래그를 삭제하면 이러한 복잡성이 줄어들어 설계를 더 간소화할 수 있습니다.</p>

<p>문제상황</p>	<p>5단계 파이프라인 CPU 설계 과정에서 데이터 해저드가 발생했습니다. 특정 명령어의 결과가 다음 명령어에서 필요할 때, 결과 값이 아직 레지스터에 쓰여지기 전에 읽으려는 문제가 발생했습니다. 예를 들어, ADD 명령어의 결과가 다음 명령어인 SUB에서 필요할 때 데이터 해저드가 발생했습니다.</p>
-------------	---

해결법	데이터 해저드를 해결하기 위해 Forwarding Unit 과 Hazard Detection Unit 을 설계했습니다. Forwarding Unit 은 EX 단계에서 필요할 데이터를 MEM 단계나 WB 단계에서 가져와 사용할 수 있게 했습니다. 또한, Hazard Detection Unit 은 데이터 해저드가 감지될 경우, 파이프라인을 스톱 시키거나 버블을 삽입하여 문제를 해결했습니다. 이를 통해 명령어 간의 데이터 의존성을 해결하고, 파이프라인의 원활한 동작을 유지할 수 있었습니다
-----	--

문제상황	파이프라인 CPU에서 분기 명령어(BEQ, BNE 등) 실행 시, 분기 여부를 결정하기 전에 이미 몇 개의 명령어가 파이프라인에 들어가 있어 제어 해저드가 발생했습니다. 이는 잘못된 명령어가 실행되는 문제를 야기했습니다.
해결법	제어 해저드를 해결하기 위해 Branch Prediction 기법과 Branch Target Buffer(BTB) 를 도입했습니다. 간단한 방식으로, 항상 분기가 일어난다고 예측하고, 예측이 틀렸을 경우 파이프라인을 플러시하여 잘못된 명령어를 제거했습니다. 또한, BEQ, BNE와 같은 분기 명령어가 발생하면, ID 단계에서 다음 명령어를 가져오지 않도록 하여 제어 해저드를 최소화했습니다. 이를 통해 제어 해저드로 인한 성능 저하를 줄일 수 있었습니다.

문제상황	<p>Time: 350000, PC: 1, num_inst: 65535, output_port: 0 ID Stage: PC: 0, Instruction: f202 EX Stage: PC: 0, ALUResult: 0 MEM Stage: PC: 0, ALUResult: 0, ReadData: zzzz WB Stage: PC: 0, ALUResult: 0, WriteData: 0 Time: 450000, PC: 2, num_inst: 65535, output_port: 0 ID Stage: PC: 1, Instruction: f302 EX Stage: PC: 0, ALUResult: 0 MEM Stage: PC: 0, ALUResult: 0, ReadData: zzzz WB Stage: PC: 0, ALUResult: 0, WriteData: 0 Time: 550000, PC: 3, num_inst: 65535, output_port: 0 ID Stage: PC: 2, Instruction: f81c EX Stage: PC: 1, ALUResult: 0 MEM Stage: PC: 0, ALUResult: 0, ReadData: zzzz WB Stage: PC: 0, ALUResult: 0, WriteData: 0 Time: 650000, PC: 4, num_inst: 0, output_port: 0 ID Stage: PC: 3, Instruction: 4623 EX Stage: PC: 2, ALUResult: 0 MEM Stage: PC: 1, ALUResult: 0, ReadData: zzzz WB Stage: PC: 0, ALUResult: 0, WriteData: 0 Time: 750000, PC: 5, num_inst: 1, output_port: 0 ID Stage: PC: 4, Instruction: f81c EX Stage: PC: 3, ALUResult: 23 MEM Stage: PC: 2, ALUResult: 0, ReadData: zzzz WB Stage: PC: 1, ALUResult: 0, WriteData: 0 Time: 850000, PC: 6, num_inst: 2, output_port: 0 ID Stage: PC: 5, Instruction: f2c4 EX Stage: PC: 4, ALUResult: 23 MEM Stage: PC: 3, ALUResult: 23, ReadData: zzzz WB Stage: PC: 2, ALUResult: 0, WriteData: 0</p> <p>코드 작성 과정에서 TESTBENCH를 통해 확인해본 결과값이 PIPELINE은 정상적으로 작동하고 메모리에 넣어뒀던 명령어 역시 올바르게 읽히는 것을 확인했지만 ALU_RESULT 값이 23으로 고정되는 오류가 발생했습니다..</p>
해결법	<p>메모리 값에서 TEST 진행 할 INSTRUCTION을 읽어와 TEST를 진행하는 방식을 선택해서 진행했고 1번 메모리에서 JMP 0X23을 통해 0X23 메모리로 이동해 계속해서 INSTRUCTION을 진행하도록 설정해놓아 ALU_Result 값이 23으로 고정된 것이라는 가정을 해봤습니다.</p> <p>pc값도 정상적으로 증가하고 instruction 또한 올바르게 읽히는 것을 보았을 때 오류는 instruction 해석부분에 있을 것이라고 판단해서 decode 부분과 opcode 부분을 다시 훑어보았습니다.</p>